Zusammenfassung DL4G

Deep Learning for Games

Maurin D. Thalmann

19. Januar 2020

Inhaltsverzeichnis

1 Sequential Games with perfect information

1.1 Finite Sequential Games

- Eine endliches Set an **Spielern**, jeder mit einem endlichen Set an möglichen **Aktionen**
- Spieler wählen ihre Aktionen **sequenziell** (einer nach dem anderen, in Zügen)
- Eine endliche Anzahl an **Zügen** wird gespielt
- Spätere Spieler **beobachten** die Züge der früheren Spieler (Perfect Recall)
- Eine **Strategie** sagt dem Spieler, welche Aktion er in seinem Zug spielen soll
- Ein **Strategieprofil** ist eine gewählte Strategie eines jeden Spielers
- Ein **Utility** oder **Payoff Function** bestimmt den Ausgang jedes Aktionprofils

1.2 Complexity Factors in Game Analysis

1. Anzahl Spieler
   - Spiele mit 4 Spielern sind schwieriger zu analysieren als solche mit 2 Spielern
2. Grösse des Suchraums
   - Bestimmt durch Anzahl gespielte Züge und Anzahl Aktionen für jeden Spieler
3. Kompetitive Spiele vs. Kooperative Spiele
   - Kompetitive Spiele involvieren Spieler mit komplett gegensätzlichen Interessen
4. Stochastische Spiele vs. Deterministische Spiele
   - Stochastische Spiele beinhalten Zufälle, bspw. Verteilung der Karten, Würfel rollen
5. Perfekte vs. imperfekte Informationsspiele
   - Imperfekte Information heisst das Spiel ist nur teilweise überwachbar, bspw. kennen wir nicht die Karten eines gegnerischen Spielers beim Poker oder Jass

1.3 Illustration of State Space Complexity

Game	**State Space (as log to base 10; 10^x)**
Tic-Tac-Toe	3
Connect-4	13
Backgammon	20
Chess	47
Go 19x19	170

- State Space beschreibt die Anzahl erlaubter Boardpositionen
- Schach / Chess hat 10^{47} verschiedene Boards, Go hat 10^{170} verschiedene Boards
- Zum Vergleich: Geschätzt sind im Universum 10^{80} Atome

1.4 Extensive Form Representation

- Sequenzielle Spiele können (im Prinzip) als Spielbäume repräsentiert werden
- Knoten sind Spielzustände/Positionen und Kanten sind Aktionen/Bewegungen
- Blätter (Leaves) bestimmen den Payoff

1.5 Game Tree Analysis - Backward Induction

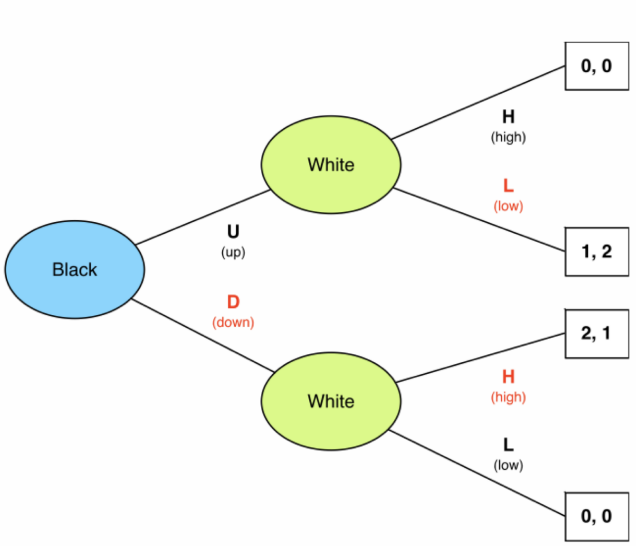


Abbildung 1: Backward Induction am Beispiel eines simplen Spielbaums

- Backward Induction ist der Lösungsalgorithmus für endliche, sequenzielle Spiele
- Im Beispiel oben hat Black den First-Mover Vorteil.
- Wenn beide Spieler perfekt spielen, endet das Spiel mit den Payoffs (2,1)
  (2 für Black, 1 für White)
- Solche Spiele immer rückwärts analysieren!

1.6 Reasoning about Finite Sequential Games

- Eine **ultra-schwache Lösung** beweist ob der erste Spieler aus der Initialposition gewinnen, verlieren oder unentschieden machen wir, in Annahme eines perfekten Spiels des Gegners
  Im Beispiel: Schwarz kann einen Gewinn forcieren und hat demnach den First-Mover Vorteil
- Eine **schwache Lösung** bietet einen Algorithmus welcher ein komplettes Spiel an perfekten Zügen aus der Initialposition offenbart, in Annahme eines perfekten Spiels des Gegners
  Schwache Lösung für das Spiel: Black spielt U, White spielt L, Black spielt D
- Eine **starke Lösung** bietet einen Algorithmus, welcher perfekte Züge aus jeder Position produzieren kann, auch wenn vorher von irgendeinem Spieler Fehler gemacht wurden.
  Starke Lösung für dieses Spiel:
  - *Algorithmus für White: wenn Black U spielt → spiel L; wenn Black D spielt → spiel H*
  - *Algorithmus für Black: Spiel D im ersten Zug; wenn White L im oberen Knoten spielt, spiel D*

A Strong Solution to Nim Algorithmus für einen perfekten Nim Bot:

1. Wenn nur ein Haufen übrig bleibt
   → Nimm alle Objekte des Haufens und hol den Preis
2. Wenn zwei Haufen mit unterschiedlicher Anzahl Objekte übrig bleiben
   → Nimm Objekte vom grösseren Haufen und mache beide gleich gross
3. Wenn zwei Haufen diesselbe Anzahl Objekte haben
   → Egal was du tust, du hast verloren (in Annahme eines perfekten Spiels des Gegners)

1.7 Zero-Sum Games

- Ein Spiel ist Zero-Sum, wenn der totale Gewinn des Siegers gleich dem totalen Verlust des Verlierers ist
  - Einen Kuchen zu schneiden ist zero-sum bspw. wenn ich ein Stück esse, ist es für dich verloren
  - Brettspiele sind zero-sum bspw. wenn der Sieger +1 erhält und der Verlierer -1
- u_1 und u_2 umschreiben die Utility-Funktion von Spieler 1 und 2, somit ist für jedes Strategiepaar s_1 und s_2 von Spieler 1 und 2 und es gilt $u_1(s_1, s_2) + u_2(s_1, s_2) = 0$, dann lässt sich sagen:

$$u_1(s_1, s_2) = -u_2(s_1, s_2)$$

Backward Induction & Minimax

- Backward Induction ist die Lösungsstrategie für endliche Spiele mit perfekter Information
- Eine einzelne Durchführung von Backward Induction aus einem Startzustand offenbart eine schwache Lösung. Wenn Backward Induction dynamisch (während des Spiels) aus jedem Zustand ausgeführt werden kann, erhalten wir eine starke Lösung.
- Wenn das Spiel zusätzlich zero-sum ist, kann Backward Induction mit dem Minimax Algorithmus implementiert werden. Minimax wird oft für Zwei-Spieler-Spiele definiert, ist aber auch für mehr Spieler erweiterbar.
- Minimax erlaubt effizientes Pruning („Ausästen“) und nahtlose Integration von Heuristiken
- Hinweis: Backward Induction kann auch für Spiele genutzt werden, die nicht zero-sum sind und kompliziertere Payoffs enthalten als eine einzelne Zahl

1.8 Minimax Algorithm

- 1928 von John von Neumann erfunden
- Beide Spieler wollen ihre respektiven Payoffs maximieren
- Weil das Spiel zero-sum ist, ist mein Gewinn = Verlust des Gegners
- Anstatt nach eigenem Gewinn zu maximieren, kann der Gewinn des Gegeners minimiert werden
- Im Spielbaum kann folgendermassen vorgegangen werden:
  - Gehört der Knoten mir, wähle die Aktion welche den Payoff maximiert
  - Gehört der Knoten dem Gegner, wähle die Aktion welche den Payoff minimiert

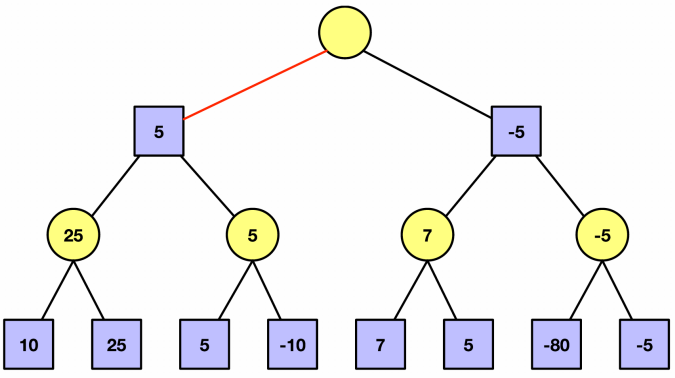


Abbildung 2: Lösungsweg eines Spielbaums mithilfe des Minimax-Algorithmus

Programming a Minimax Bot

- Minimax implementiert Backward Induction für zero-sum Spiele. Dank der vereinfachenden zero-sum Eigenschaft können einige Tricks angewandt werden, um einen herusitischen Algorithmus zu erhalten:
  - Minimax nur bis zu einer limitierten Tiefe, bspw. 5 Runden vorausschauen und dann stoppen. Wie tief man gehen kann ist abhängig von den Spielregeln (Anzahl mögliche Züge), Effizienz der Implementation und Rechenleistung.

– Da die Tiefe limitiert ist, werden die Blattknoten nicht erreicht und wir kennen die echten Payoffs nicht. Wir müssen eine Heuristik erfinden, um die Situation abzuschätzen an welcher wir stoppen. Je besser die Heuristik, desto besser der Bot.

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, FALSE)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, depth - 1, TRUE)
            bestValue := min(bestValue, val)
        return bestValue
```

Abbildung 3: Pseudocode eines Minimax-Algorithmus mit limitierter Tiefe und Heuristik

1.9 Search Tree Pruning

- Es müssen nicht alle Knoten abgelaufen werden, um die optimale Strategie zu finden.
- Wir „stutzen“ (prunen) Sub-Bäume, welche keine bessere Lösung beinhalten können und demnach nicht besucht werden müssen.
- Dazu enthält der Algorithmus zwei Parameter:
  (Vorfahren sind alle Knoten auf dem Weg zwischen dem aktuellen und dem Wurzelknoten)
  – α ist der höchste Wert aller MAX-Vorfahren eines MIN Knoten
  – β ist der tiefste Wert alles MIN-Vorfahren eines MAX Knoten
- Der Algorithmus Alpha-Beta Pruning aktualisiert diese beiden Parameter im Minimax-Prozess und schneidet nicht besuchte Sub-Bäume ab, sobald er weiss dass die Werte aus diesem Sub-Baum den Wert α nicht überbieten oder β nicht unterbieten können.

Alpha-Beta Pruning Rules

- Alpha (α) ist der minimale Score, welcher dem maximierenden Spieler versichert werden kann
- Beta (β) ist der maximale Score, welcher dem minimierenden Spieler versichert werden kann
- Daraus lassen sich folgende beiden Regeln schliessen:
  – **Regel 1**: Schneide ab, sobald der aktuelle Wert eines MIN Knoten kleiner ist als α
  – **Regel 2**: Schneide ab, sobald der aktuelle Wert eines MAX Knoten grösser ist als β

1.10 Illustrations for Alpha-Beta Pruning

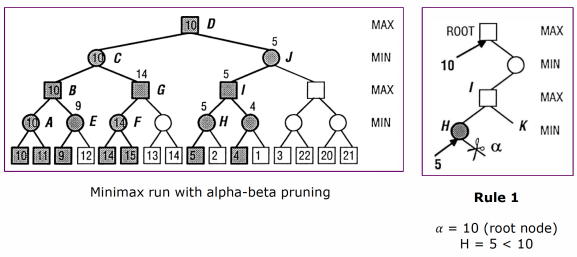


Abbildung 4: Illustration der Durchführung der 1. Regel des Alpha-Beta Pruning

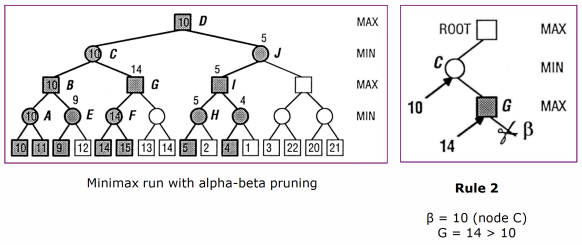


Abbildung 5: Illustration der Durchführung der 2. Regel des Alpha-Beta Pruning

```
function alphabeta(node, depth, α, β, maximizingPlayer)
     if depth = 0 or node is a terminal node
         return the heuristic value of node
     if maximizingPlayer
         for each child of node
             α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
             if β ≤ α
                 break (* β cut-off *)
         return α
     else
         for each child of node
             β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
             if β ≤ α
                 break (* α cut-off *)
         return β
```

Abbildung 6: Pseudocode eines Minimax mit limitierter Tiefe mithilfe von Alpha-Beta Pruning

Speed-Up of Alpha-Beta Pruning

- In einem Spielbaum mit Tiefe m mit b möglichen Aktionen bei jedem Knoten ist die Zeitkomplexität des Minimax O(b^m) bzw.. es gibt b^m Blattknoten
- Im Idealfall benötigt Alphe-Beta Pruning nur O($b^{m/2}$) = O($(\sqrt{b})^m$). Dies korrespondiert zu einer Reduzierung des Branching-Faktors von b zu $\sqrt{b}$, bspw. bei Schach bedeutet dies 6 mögliche Aktionen bei jedem Knoten (anstelle von 35)
- Um diesen maximalen Speed-Up zu erreichen, müssen die verschiedenen States in gescheiter Anordnung erforscht werden, was jedoch problemspezifisch ist.

2 Monte Carlo Tree Search

2.1 Random Walks

- Suchräume sind meist zu gross für eine vollständige Suche
- Minimax soll bei einer bestimmten Baum-Tiefe stoppen und raten (mit Heuristik)
- Monte Carlo Tree Search ist ein anderes Vorgehen:
  Monte Carlo Tree Search führt Random Walks durch, um möglichst viel des Suchbaums in einem vorbestimmten Zeitraum abzusuchen. Danach wird der vielversprechendste Zug gespielt.

2.2 The 4 Phases in Monte Carlo Tree Search

1. **Selection**
   - Starte beim Wurzelknoten `R` und wähle fortlaufend Kinderknoten
   - Stoppe, wenn du einen Knoten erreichst, der noch nicht komplett erweitert/erforscht wurde
   - Benötigt ein Kriterum für die Auswahl der Kinderknoten, sogennante *tree policy*
2. **Expansion**
   - Wenn das Zeitlimit `L` das Spiel beendet, gib die Payoffs zurück
   - Sonst, wähle eine unerforschte Aktion und kreiere einen Knoten `C` für diese
3. **Simulation**
   - Simuliere ein Weiterspielen von Knoten `C` aus, mithilfe einer *default policy*
   - Im simpelsten Fall, spiele einfach bis zu irgendeinem Ende mit zufälligen Zügen
4. **Backpropagation**
   - Aktualisiere die gespeicherten Informationen in jedem Knoten von `C` zurück bis zu `R`
   - MCTS erwartet einen Payoff in [0,1]

2.3 MCTS for Tic-Tac-Toe

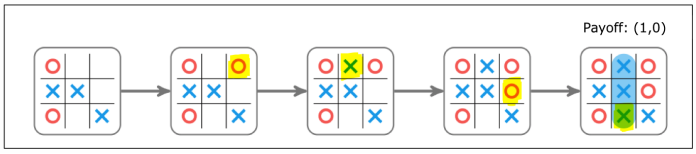


Abbildung 7: Zufälliger Durchlauf eines Runde Tic-Tac-Toe

- Es gibt Punkte für einen Gewinn (1), unenentschieden (0.5) und Verlust (0)
- MCTS ist nicht limitiert auf zero-sum Spiele
- Payoffs werden durch Vektoren repräsentiert
- Die Simulation endet mit einem Gewinn für Spieler X mit Payoff +1 → Payoff-Vektor (1,0) (Spieler Y hat in dieser Simulation also einen Payoff von 0)

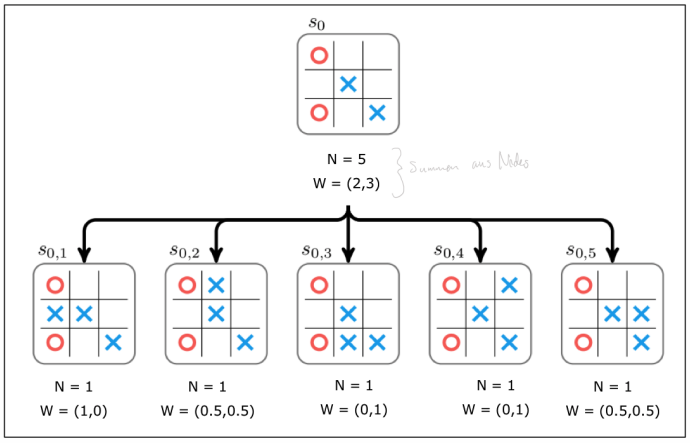
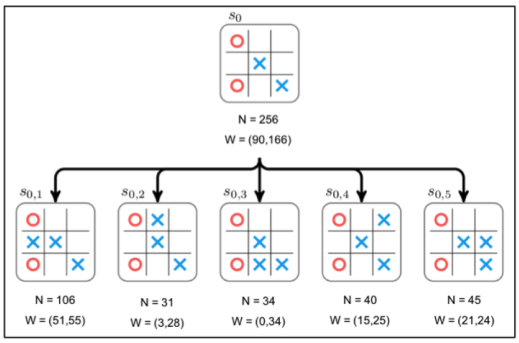


Abbildung 8: übersicht der Simulationen eines Tic-Tac-Toe Spiels

- `N` speichert die Anzahl Simulationen, die vom Knoten s_0 aus gestartet wurden
- `W` sind die akkumulierten Payoff-Vektoren (eine Komponente für jeden Spieler)
- Backpropagation kalkuliert lediglich die Summe von $\texttt{N}^*$ und `W`
- **Wichtig**: Ist der Wurzelknoten ebenfalls auch ein Kindesknoten, muss sein Payoff nach oben ebenfalls hinzugerechnet werden

2.4 Selection Policy: Which Node to Choose



Node	Mean Payoff
$s_{0,1}$	51/106 = 0.481
$s_{0,2}$	3/31 = 0.097
$s_{0,3}$	0/34 = 0.000
$s_{0,4}$	15/40 = 0.375
$s_{0,5}$	21/45 = 0.467

Abbildung 9: Durchschnittliche Payoffs der durchlaufenen Simulationen mit MCTS

- **Exploitation**
  Knoten $s_{0,1}$ hat den höchsten durchschnittlichen Payoff bzw. basierend auf aktuell verfügbaren Informationen, maximiert dieser Knoten meinen erwarteten Gewinn
- **Exploration**
  Knoten $s_{0,4}$ wurde nur 40 mal probiert, wie kann ich sicher sein dass der Score tiefer bleibt als bei $s_{0,1}$, auch wenn ich jetzt noch 66 mal spielen würde?

2.5 Example of Multi-Armed Bandits

- Es gibt eine Reihe an Slotmaschinen mit verschiedenen (unbekannten) Auszahlungswahrscheinlichkeiten und -mengen
- Man hat 1000 Münzen und möchte den erwarteten Gewinn erhöhen
- Idealerweise würde man immer an der Maschine mit dem grössten erwarteten Gewinn spielen
- Unglücklicherweise weiss man jedoch nicht, welche Maschine dafür am besten ist
- Es ist keine Person da, welcher man zuschauen und mehr über die Maschinen erfahren kann

Wie wählt man nun die beste Strategie in dieser Situation?

2.6 UCB1: Upper Confidence Bound

- Am besten Exploration und Exploitation ausbalancieren
  - **Exploration**: Spiele alle Maschinen um möglichst viele Informationen zu sammeln
  - **Exploitation**: Spiele die beobachtet beste Maschine um den erwarteten Gewinn zu maximieren
- UCB1 bietet die beste Balance zwischen Exploration und Exploitation, es gibt quasi ein statistisches Vertrauensintervall für jede Maschine aus
  - Parameter $c \geq 0$ kontrolliert den Trade-Off zwischen Exploitation (tiefes c) und Exploration (hohes c)

$$U_i = \frac{W_i}{N_i} + c\sqrt{\frac{\ln N_p}{N_i}}$$

$\frac{W_i}{N_i}$ **Exploitation**, der durchsch. Gewinn für Maschine i ($\frac{W(Gewinne)}{N(Versuche)}$)

$\sqrt{\frac{\ln N_p}{N_i}}$ **Exploration**

N_p Wie oft haben wir insgesamt gespielt (wie viele Münzen wurden verbraucht)

N_i Wie oft wurde Maschine i ausprobiert

How to play with UCB1 Für jede der 1000 Münzen...

- Kalkuliere das UCB1 (U_i) von jeder Maschine i
- Spiele an der Maschine mit dem höchsten Upper Bound (U_i)
- Wähle entweder zufällig oder in numerischer Ordnung
- Solange man dies tut, wird der beobachtete Mittelwert für die Maschine sich verschieben und das Vertrauensintervall wird schmaler, aber die Intervalle aller anderen Maschinen werden breiter. Der Upper Bound einer anderen Maschine wird womöglich grösser als jener der aktuellen Maschine und man wird zu dieser Maschine wechseln.
- Bei Berechnung des UCB1 muss die Vektorkomponente für den aktuellen Spieler gewählt werden! Bsp. UCB1($s_{0,1}$) = 0.71, UCB1($s_{0,2}$) = ...

The final Move

- Wenn die Zeit vorüber ist, spiele die erste Aktion mit der **höchsten Anzahl an Besuchen** N!
  - Es benötigt keine Exploration wenn die finale Aktion ausgewählt werden soll
  - Deshalb ist die Aktion mit dem höchsten UCB1 Score in der letzten Runde nicht zwingend die beste Wahl
  - Die Aktion, welche am meisten gewählt wurde, gab meistens auch den höchsten UCB1 Score

2.7 Minimax vs. Monte Carlo Tree Search

- Beide Algorithmen setzen perfekte Information voraus
- Minimax ist nur für 2-Spieler zero-sum Spiele anwendbar
- MCTS funktioniert für jedes perfekte-Informationsspiel
- Minimax optimiert Payoffs; MCTS optimiert einen Exploitation-Exploration Trade-Off
- In Kontrast zu Minimax, ist MCTS ein „Anytime“ Algorithmus
- Monte Carlo Bäume sind asymmetrisch, Minimax Bäume sind symmetrisch

2.8 MCTS for Zero-Sum Games

- Im Fall von 2 Spielern, Payoff Vektoren sind von der Form (W, $N - W$)
- Spieler 1 maximiert W, Spieler 2 maximiert $N - W$ (ignoriert den Exploitation-Teil in UCB1)
- Anders gesagt, könnte Spieler 2 $-W$ minimieren
- Wir können nur die Rewards W für Spieler 1 speichern und die Zeichen für Spieler 2 „flippen“
- Daraus entsteht eine Minimax-ähnliche Variante von MCTS

Convergence to Optimal Play

- Mit genügend Ressourcen konvergiert MCTS zu Minimax
- UCB1 hat logarithmisches Bereuen (*logarithmic regret*), weil nur bestimmte Zeit zur Verfügung steht
- Die Fehlerwahrscheinlichkeit an der Wurzel des Baums konvergiert zu null in polynomialer Rate wenn die Anzahl simulierter Spiele in Richtung unendlich ansteigt. Dieser Beweist bedeutet, unter Annahme von genug Zeit, dass UCB1 der MCTS erlaubt zum Minimax-Baum zu konvergieren und ist demnach optimal.

3 Information Sets

4 Supervised Machine Learning

5 Neuronal Networks

6 Deep Neuronal Networks

7 Convolutional Neuronal Networks